A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android

David Barrera
dbarrera@ccsl.carleton.ca

H. Güneş Kayacık
kayacik@ccsl.carleton.ca

P.C. van Oorschot
paulv@scs.carleton.ca

Anil Somayaji
soma@scs.carleton.ca

School of Computer Science, Carleton University
Ottawa, ON, Canada

ABSTRACT

Permission-based security models provide controlled access to various system resources. The expressiveness of the permission set plays an important role in providing the right level of granularity in access control. In this work, we present a methodology for the empirical analysis of permission-based security models which makes novel use of the Self-Organizing Map (SOM) algorithm of Kohonen (2001). While the proposed methodology may be applicable to a wide range of architectures, we analyze 1,100 Android applications as a case study. Our methodology is of independent interest for visualization of permission-based systems beyond our present Android-specific empirical analysis. We offer some discussion identifying potential points of improvement for the Android permission model, attempting to increase expressiveness where needed without increasing the total number of permissions or overall complexity.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*Connectionism and neural nets*; D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*

General Terms

Security, Experimentation

Keywords

Access control, self-organizing maps, permission-based security, smartphone operating systems, visualization

1. INTRODUCTION

Access control lists (ACLs) and permission-based security models allow administrators and operating systems to restrict actions on specific resources. In practice, designing and configuring ACLs (particularly those with a large number of configuration parameters) is a complicated task. More specifically, reaching a balance between the detailed expressiveness of permissions and the usability of the system is not trivial, especially when a system will be used by novices and experts alike.

One of the main problems with ACLs and permission models in general is that they are typically not designed by the users who will ultimately use the system, but rather by developers or administrators who may not always forsee all possible use cases. While some argue that the problem with these permission-based systems is that they are not designed with usability in mind [11], we believe that in addition to the usability concerns, there is not a clear understanding of how these systems are used in practice, leading security experts to blindly attempt to make them better without knowing where to start.

While there are many widely deployed systems which use permissions (some are discussed in Section 2.2), we focus on the empirical analysis of the permission model included in Android OS [1]. Android is a newcomer to the smartphone industry and in just a few years of existence has managed to obtain significant media attention, market share, and developer base. Android uses ACLs extensively to mediate inter-process communication (IPC) and to control access to special functionality on the device (e.g., GPS receiver, text messages, vibrator, etc.). Android developers must request permission to use these special features in a standard format which is parsed at install time. The OS is then responsible for allowing or denying use of specific resources at run time. The permission model used in Android has many advantages and can be effective in preventing malware while also informing users what applications are capable of doing once installed.

The main objectives of our empirical analysis are: (1) to investigate how the permission-based system in Android is used in practice (e.g., whether the design expectations meet the real-world usage characteristics) and (2) to identify the strengths and limitations of the current implementation. We believe such analysis can reveal interesting usage patterns, particularly when the permission-based system is being used by a wide spectrum of users with varying degrees of expertise. As of July 2010, there are over 80,000 applications available for Android [2], many of which are free and written by both large software development companies and hobbyists. Also, the Android Market is not controlled as tightly as other mobile application stores [5]. This implies the applications may exhibit more variety in terms of requested permissions along with other behavioral characteristics which might not occur in a closed environment.

Contributions. Our main contribution is a novel methodology for exploring and empirically analyzing permission-based models. In this paper, we employ our methodology for the analysis of 1,100 applications written for the Android OS. Using the Self-Organizing Map (SOM) algorithm [16], we identify trends in how developers of these applications use the Android permissions model. We find that while Android has a large number of permissions restricting access to advanced functionality on devices, only a small number of these permissions are actively used by developers. Our analysis identifies permissions that are overly broad (i.e., controlling access to a large set of features). Furthermore we identify application clusters based on requested permissions, and extract the prominent permissions within each cluster. Our empirical observations provide a basis for possible enhancements to the Android permission model.

The remainder of this paper is structured as follows. Section 2 presents background on permission-based security architectures, provides examples of some currently deployed permission-based systems and discusses related work. Section 3 describes the Android operating system and its novel permission model. The dataset used in our case study is also covered in this section. In Section 4 we discuss our methodology based on the Self-Organizing Map algorithm. Section 5 covers the results of our analysis and discusses the generated visualizations. Section 6 summarizes key findings and suggests points for improvement in Android. We conclude in Section 7.

2. BACKGROUND

Access control systems have existed for a long time [17]. In its basic form, a security system based on access control lists allows a *subject* to perform an *action* (e.g., read, write, run) on an *object* (e.g., a file) only if the *subject* has been assigned the necessary permissions. Permissions are usually defined ahead of time by an administrator or the object's owner. Basic file system permissions on POSIX-compliant systems [12] are the traditional example of ACL-based security since objects – in this case, files – can be read, written or executed either by the owner of the file, users in the same group as the owner, and/or everyone else. More sophisticated ACL-based systems allow the specification of a complex policy to control more parameters of how an object can be accessed.

We use the term *permission-based security* to refer to a subset of ACL-based systems in which the *action* doesn't change (i.e., there is only one possible action to allow or deny on an object). This would be similar to having multiple ACLs per object, where each ACL only restricts access to one action. We note that reducing the allowable actions to one does not necessarily make the system easier to understand or configure. For example, in the Android permission model, developers implement finer level granularity by defining separate permissions for read and write actions. This implies that, compared to general ACLs, the permission hierarchy is flat and has limited sense of grouping.

2.1 Permission-Based Security Examples

An example of a permission-based security model is Google's Android OS for mobile devices. Android requires that developers declare in a manifest a list of permissions which the user must accept prior to installing an application. Android uses this permission model to restrict access to advanced or dangerous functionality on the device [14]. The user decides whether or not to allow an application to be installed based on the list of permissions included by the developer. We describe the Android security architecture in detail in Section 3.

Similar to Android OS, the Google Chrome web browser uses a permission-based architecture in its extension system [4]. Extension developers create a manifest where specific functionality (e.g., reading bookmarks, opening tabs, contacting specific domains) required by the extension can be requested. The manifest is read at extension install time to better inform the user of what the extension is capable of doing, and reduce the privileges that extensions are given [10]. In contrast, Firefox extensions, which do not have this permission architecture, run all extension code with the same OS-level privileges as the browser itself.

A third example of a currently deployed permission-based architecture is the Blackberry platform from Research In Motion (RIM). Blackberry applications written in Java must be cryptographically signed in order to gain access to advanced functionality (known as Blackberry APIs with controlled access) such as reading phone logs, making phone calls or modifying system settings [3]. The Blackberry OS enforces through signature validation that an application has been granted permissions to access the controlled APIs.

2.2 Related Work

Enck et al. [13] describe the design and implementation of a framework to detect potentially malicious applications based on permissions requested by Android applications. The framework reads the declared permissions of an application at install time and compares it against a set of rules deemed to represent dangerous behaviour. For example, an application that requests access to reading phone state, record audio from the microphone, and access to the Internet could send recorded phone conversations to a remote location. The framework enables applications that don't declare (known) dangerous permission combinations to be installed automatically, and defers the authorization to install applications that do to the user.

Ontang et al. [18] present a fine-grained access control policy infrastructure for protecting applications. Their proposal extends the current Android permission model by allowing permission statements to express more detail. For example, rather than simply allowing an application to send IPC messages to another based on permission labels, context can be added to specify requirements for configurations or software versions. The authors highlight that there are real-world use cases for a more complex policy language, particularly because untrusted third-party applications frequently interact on Android.

On the topic of analysis of permission-based architectures, Barth et al. [10] analyzed 25 browser extensions for Firefox and identified that 78% are given more privileges than necessary, increasing the attack surface on these feature-enhancing add-ons. The analysis lead the authors to the design of a permission-based system for browser extensions in Google Chrome. The system controls access to bookmarks, tabs, and domains available to a particular extension. Investigating the usability of permission-based architectures, Reeder et al. [19] developed a framework for displaying and editing file permissions on a Windows operating system. They employed a matrix-based visualization called expandable grid, which provides a conceptual visualization of file permissions in a graphical format. Their user studies showed this grid visualization allows users to complete tasks quickly and more accurately.

Bearing similarities to our work, but not in methodology or application, Smetters et al. [20] conducted a study of permission-based architectures, particularly access control lists for document sharing within an organization. Various data mining techniques were utilized to understand how employees used access control lists. Smetters et al. argued that to find the appropriate balance between control and complexity in a permission-based architecture, it is important to determine what level of control users need by analyzing how users interact with the architecture in practice. In this paper, we analyze the real world use of Android permissions through an empirical analysis.

3. ANDROID PERMISSION MODEL

We review the Android operating system and its permission-based security model. We also discuss the dataset used for our analysis and highlight some initial observations made prior to applying our data mining methodology.

3.1 Android

Android is middleware for mobile devices that is built on top of Linux. Currently mainly deployed on smartphones, the Android platform is quickly gaining market share and due to its open source nature, has been ported to other devices such as laptops, tablets and ebook readers. Android has a strong focus on security and attempts to address some of the shortcomings of other mobile operating systems. Android applications are written in Java syntax and each run in a custom virtual machine known as Dalvik, a light-weight replacement for the standard Java Virtual Machine. The effect of running each application inside a virtual machine is extensive process isolation which in at least one instance [6] has prevented an exploit in an application to also impact other parts of the OS. Isolation is further enforced by each application being installed as its own user and group ID [14].

Applications written for Android can be distributed to end users directly through a developer's web site, or through the on-device application store known as the Android Market. The Android Market offers a central location where developers can submit their applications and, with minimal interaction from Google, reach end user devices. This differs from the Apple iTunes App Store where all applications must pass through a vetting process [5] (performed by Apple in a closed manner) before reaching consumers. Android Market applications are not always inspected upon submission, allowing malicious application developers to quickly get their applications onto end user devices. With this security concern in mind, Android has been designed to isolate third party applications from each other, as well as to protect the operating system and users from malicious applications.

3.2 Android Permissions

At the core of the Android security security model is a permission-based system that by default denies access to features or functionality that could negatively impact the user experience, the system, or other applications installed on the device. Examples of these features are sending messages or making phone calls, which may incur monetary cost to the user; keeping the device screen on or accessing the vibrator, which could result in battery drain; and reading the user's address book which could result in privacy violations.

To make use of the restricted functionality (which could potentially be dangerous if used in combination with other features, or in a different way than intended by the Android OS designers), Android requires application developers to declare which of the restricted features are intended to be used by their application. Failure to declare a particular permission will result in the related system call or inter-process communication being denied[1] by Android. There are currently 110 items of functionality which are identified as requiring an explicit permission in order for Android to grant access [8]. These permissions control access to network and GPS functions, personal information, system hardware and settings, and many other device features. However, Android is designed such that any third party application can define new functionality (e.g., through an API) and make that specific functionality available to other applications based on *developer-defined* permissions. In the case of developer-defined (also known as *self-defined*) permissions, Android enforces that both the caller and callee applications have matching permissions (i.e., the callee defined the permissions using the *permission* tag in its manifest, and the caller requested the permissions using *uses-permission*) to allow the IPC to take place.

Every application (including system applications such as the phone or calendar applications) written for the Android

[1]The actual interaction is mediated by IBinder which is a Linux Kernel module.

platform must include an XML-formatted file named *AndroidManifest.xml*. This essential part of the Android security model contains (along with other metadata such as minimum OS version requirements) the permission declarations that the application is requesting access to [7]. Permissions are declared in the manifest using the *uses-permission* tag followed by a common namespace (usually `android.permission.*` for Google defined permissions, and expressed in this paper as `a.p.*`). Self-declared permissions which other applications can request are labeled with the *permission* tag. The Android manifest contains entries which can be automatically generated by the developer environment but some fields, specifically those related to permission declarations, must be manually entered. Manual permission declaration can lead to application developers over-declaring or erroneously declaring permissions that don't exist, as explained in Section 3.4

Permissions are enforced by Android at runtime, but must be accepted by the user at install time. When users install a new application in Android (regardless of how the application was obtained), they are prompted to accept or deny the permissions requested by the application. Permissions are also described in a more user friendly language at install time. These descriptions attempt to give a brief, technical explanation of the permission, but do not disclose what the developer intends to use access to those resources for.

3.3 Dataset

The dataset used for our empirical study consists of 1,100 applications, the top[2] 50 free applications downloaded in December 2009 from each of the 22 categories in the Android Market. In the Market, Google makes a distinction between applications and games. Both of these main categories are further subdivided: the games category into 4 sub-categories (Brain and Puzzle, Arcade and Action, Cards and Casino, and Casual); the applications into 18 sub-categories (Comics, Communication, Entertainment, Finance, Health, Lifestyle, Multimedia, News and Weather, Productivity, Reference, Shopping, Social, Sports, Themes, Tools, Travel, Demo, and Software libraries).

Applications in our dataset were obtained in standard ZIP or ZIP-compatible Android Packages (APK). For each application, we used the Android Asset Packaging Tool to extract the manifest and read all XML entries of type *uses-permission* (i.e., permissions that are being requested, not newly defined). Each application is then represented as a bit vector $x = [x_1, x_2, \ldots, x_j]^T \in \{0,1\}^j$, in which x_j denotes whether the permission j is requested. Representing applications this way allows us to employ Self-Organizing Maps (SOM) for our analysis and enables us to utilize a suitable distance metric to express similarities between applications.

Table 1 lists each of the 22 categories in the Android Market and provides an aggregate count of all the unique permission labels requested by each application in a given category (i.e., if 5 applications in a category requested access to system configuration, it is only counted once). The fourth column presents the average number of permissions requested by applications in each category. The Communication category is by far the most diverse, with 62 permissions requested. This category also had the highest number of permissions per application, with one application (Handcent SMS) requesting 22 different permissions. The Themes category was the least diverse containing 49 applications which requested no additional permissions and one which requested access to the Internet.

[2]Top applications on the Android Market are believed to be ranked by Google using a combination of number of downloads and aggregate user rating of the application.

Type	Category	Permissions	Avg. Perms.
App.	Comics	9	0.98
	Communication	62	6.72
	Demo	16	1.46
	Entertainment	21	2.86
	Finance	21	1.84
	Health	15	1.50
	Libraries	40	1.36
	Lifestyle	45	3.42
	Multimedia	34	3.60
	News	22	3.62
	Productivity	52	3.98
	Reference	21	2.20
	Shopping	35	4.08
	Social	37	4.52
	Sports	17	2.20
	Themes	1	0.02
	Tools	49	3.88
	Travel	40	3.74
Games	Arcade	7	1.74
	Casino	15	2.30
	Casual	14	2.00
	Puzzle	10	1.60

Table 1: Total number of permissions requested for each of the 22 categories in the Android Market.

Our dataset includes 119 distinct permission requests, out of which 38 (31.93%) were to self-defined permissions, and the remainder were to Android permissions in the `android.permission.*` namespace. Figure 1 shows the distribution of the requested permissions, in which the y-axis denotes the number of applications that request the permission and the x-axis denotes the permission index. Permissions are ordered according to the their request counts. The `a.p.INTERNET` permission (indexed as 1) was requested by 686 applications (62.36% of all applications). `a.p.RECEIVE_BOOT_COMPLETED` (indexed as 10), which enables an application to start at system boot, was requested by 63 applications (5.72% of the total). The distribution of permissions in Figure 1 demonstrates that the frequency of permission requests decays rapidly, and there is a long tail of permissions that are only requested by one or two applications in the dataset.

Dataset Limitations. Our dataset contains applications from a large number of developers in a broad range of categories. While we expect this dataset to be representative and diverse, selecting the top-ranked applications might have the effect of filtering out applications that are either poorly written and/or malicious. Our dataset contains no known malware, and could be biased towards any trends in developer activity in December 2009. We believe, however, that our dataset reflects real-world usage trends since previous analysis we carried out on a smaller and earlier dataset gave similar results.

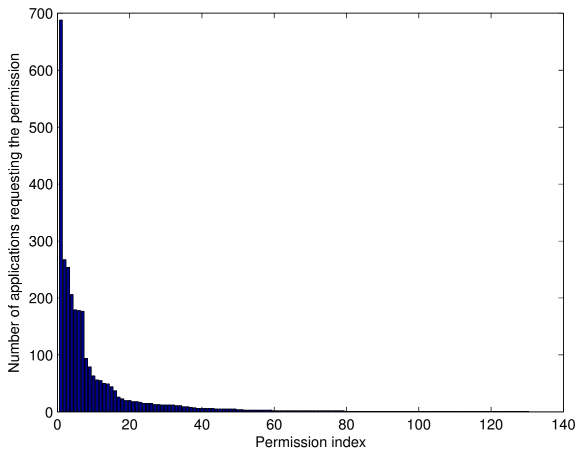

Figure 1: Permission labels exponential decay

3.4 Observations

During our dataset analysis, we identified several cases where developers requested the same permission twice in their application. This is a developer error, since adding the permission more than once does not have any added benefit. The duplicate permission error was not only seen in lesser known applications such as Quick Uninstaller (which requested the `a.p.READ_PHONE_STATE` and `a.p.INTERNET` permissions twice), but also on popular applications such as Fring (which requested the `a.p.INTERNET` permission twice). This error is likely due to limitations of the IDE, and/or the different ways in which Android maps system calls to permissions.

Another observation is that applications request permissions that do not exist. For example, the Txeet application requested access to the `a.p.ACCESS_COURSE_LOCATION` permission. The developer was likely attempting to declare the `a.p.ACCESS_COARSE_LOCATION`. If the Txeet application attempts to make use of coarse location features, Android will throw a security exception and deny access to location data since the permission is incorrect. Applications also requested `a.p.ACCESS_ASSISTED_GPS` and `a.p.ACCESS_CELL_ID`, which have been superseded by the coarse and fine location permissions since even before Android 1.0. These permissions appear to be requested in parallel to their correct equivalent permissions in more recent Android releases.

We also found an application that requests `a.p.BRICK`, which theoretically allows an application to completely disable the device. While the `a.p.BRICK` permission controls access to calls that can disable the device, third party applications cannot make use of this permission because Android restricts use of related function calls to applications signed by the same private key as the running Android OS. These permissions are known as *signature permissions*.

4. METHODOLOGY

In a typical permission-based architecture, numerous permissions are at the user's disposal. In our work, we aim to understand how the Android permission model is used in practice and to determine its shortcomings. While various manual analysis and data mining techniques are no doubt applicable, in our work, we make novel use of the Self-Organizing Map (SOM) algorithm [16], which presents a simplified, relational view of a highly complex dataset by preserving proximity relationships. The characteristics that make SOM suitable for the analysis are that (1) SOM provides a 2-dimensional visualization of the high dimensional data, and (2) the component analysis of SOM can identify correlation between permissions.

Our methodology allows us to gain insights on how the developers use the given permission model in practice and highlights the strengths of the permission model as well as it shortcomings. We note that, although the case study focuses on Android, our empirical analysis is suitable for various other permission-based architectures, as long as the applications are represented as a bit string of permissions, as discussed in Section 3.3.

4.1 Self-Organizing Maps

The Self-Organizing Map (SOM) is a type of neural network algorithm, which employs unsupervised learning (i.e., without requiring any labels) to produce a typically 2-dimensional, discretized representation of a high dimensional input space. SOM aims to summarize complex datasets while preserving the topological properties of the input space. SOM consists of neurons, which have the same dimensionality as the input space. The neurons are typically arranged in a rectangular or a hexagonal grid. SOM neurons can be considered as pointers in the input space, in which more neurons point to regions with high concentration of inputs.

The training is competitive. Specifically, when an input is presented, its Euclidean distance to each SOM neuron is calculated. The neuron with the minimum distance – the best matching neuron – is identified. The weight values of the best matching neuron and its adjacent neurons are adjusted towards the input vector. Updating neurons this way associates them with groups of patterns in the input dataset. Training is repeated for each input until the input dataset is processed several times.

The training algorithm can be summarized in four basic steps. Step 1 initializes the SOM before training. Step 2 determines the best matching neuron, which is the shortest Euclidean distance to the input pattern. Step 3 involves adjusting the best matching neuron and its neighbors so that the region surrounding the best matching neuron becomes closer to the input pattern. This causes SOM to minimize the distance between the updated region and the input pattern. This training process continues until all input vectors are processed. The convergence criterion utilized in SOM is in terms of *epochs*, which define how many times all input vectors should be fed to the SOM for training. Details of the SOM algorithm follow:

Step 1: Initialize neuron weights $w_i = [w_{i1}, w_{i2}, \ldots, w_{ij}]^T \in \Re^j$. In our work, neuron weights are initialized with random numbers.

Step 2: Present an input pattern $x = [x_1, x_2, \ldots, x_j]^T \in \Re^j$. In this work, each input pattern corresponds to an application in which the permissions are expressed in the form of a bit string. For example, an application is represented as the bit string $[1, 1, 1, 0, 0]^T$ if it requests permissions 1, 2, 3 but not 4 and 5. Calculate the distance between pattern x, and each neuron weight w_i, and therefore identify the winning neuron or best matching neuron c as follows:

$$\|x - w_c\| = \min_i\{\|x - w_i\|\} \tag{1}$$

In our work, we employed Euclidian distance as the distance metric, normalized to the range $[0, 1]$.

Step 3: Adjust the weights of winning neuron c and all neighbor units

$$w_i(t+1) = w_i(t) + h_{ci}(t)[x(t) - w_i(t)] \tag{2}$$

where i is the index of the neighbor neuron and t is an integer, the discrete time coordinate. The neighborhood kernel $h_{ci}(t)$ is a function of time and the distance between neighbor neuron i and winning neuron c. $h_{ci}(t)$ defines the region of influence that the input pattern has on the SOM and consists of two components [22]: the neighborhood function $h(\|\cdot\|, t)$ and the learning rate function $\alpha(t)$, in Equation 3:

$$h_{ci}(t) = h(\|r_c - r_i\|, t)\alpha(t) \tag{3}$$

where r is the location of the neuron on two dimensional map grid. In our work, we used Gaussian Neighborhood Function. The learning rate function $\alpha(t)$ is a decreasing function of time. The final form of the neighborhood kernel with Gaussian function is

$$h_{ci}(t) = \exp\left(-\frac{\|r_c - r_i\|^2}{2\sigma^2(t)}\right)\alpha(t) \tag{4}$$

where $\sigma(t)$ defines the width of the kernel.

Step 4: Repeat steps 2 - 3 until the convergence criterion is satisfied.

The training is conducted in two stages. In rough training, the learning rate (i.e., $\alpha(t)$) is set to a higher value, hence has the potential to cause greater changes in SOM. On the other hand, in fine tuning, the learning rate is reduced to facilitate incremental changes in neurons. Training parameters of the SOM employed in this paper are summarized in Table 2. In this paper, SOM is used for data exploration. Thus, following the common practices of exploratory data analysis, training parameters are empirically tested before producing the final visualizations.

Parameter	**Rough Training**	**Fine Tuning**
Initial α	0.5	0.05
α decay scheme	inverse_t	
Epoch Limit	2,000	
Map Size	10 x 10	
Initial Neighborhood Size	3	1
Neighborhood Function	Gaussian	
Neighborhood Relation	Hexagonal	

Table 2: SOM training parameters

5. RESULTS

One of the advantages of SOM over other unsupervised learning techniques such as typical clustering algorithms [15] is its ability to create a 2-dimensional visualization of high dimensional cluster structures. As discussed in Section 4.1, each SOM neuron can be considered as a 'pointer' in permission space. Thus, applications can be assigned to the nearest neuron, effectively clustering the applications requesting similar permissions into the same neighborhood. To visualize the cluster structure of high dimensional weight vectors of SOM neurons, a graphic display called U-matrix [21] is used.

Although the training is unsupervised, after training, labels from the training data (i.e., the application categories) are used to automatically assign labels to neurons. To label SOM neurons, a winner-take-all scheme is employed. The labeling scheme makes use of the application categories of the 1,100 Android applications. Post-training, the input data is fed to SOM to determine the best matching neuron for each application and a record of the best matching neuron for each application is maintained. The most predominant category for each neuron determines the label. In other words, a neuron is labeled as Communication if the majority of applications, for which the neuron was the best matching, was from the Communication category.

Assigning labels to SOM neurons helps us to build a 2-dimensional visualization which highlights the similarities between applications from different categories. Coupled with the U-matrix visualization, the labeled map provides a visual representation of applications where similar applications (in terms of requested permissions since they can be from different categories) are placed in close vicinity.

The U-matrix representation of the SOM for Android permissions is shown in Figure 2. This representation employs a heat colormap to show the distances between weight vectors of the neurons. The heat colormap ranges from black through shades of red and yellow to white (black to grey to white, when printed in grayscale), where white implies 'hot' or high values and black implies 'cold' or low values. Therefore, if the distance between neighboring neurons is small, the region is drawn with dark colors. Conversely, if the distance between neighboring neurons is large, a light shade is used. The U-matrix representation employs extra hexagons between neurons to show the topology of the clusters. For example, in Figure 2, the top left hexagon represents the neuron associated with the Travel category, in Figure 3. However, the adjacent neuron associated with the Lifestyle category (see Figure 3) is the third hexagon from the top left since the hexagon between the two neurons is introduced to express distance between them.

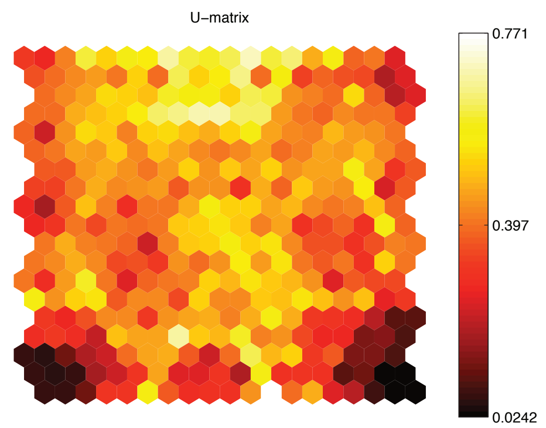

Figure 2: U-matrix representation of the SOM for Android permissions, with additional hexagons to express the distance between neurons. The scale shows the normalized Euclidean Distance [0,1].

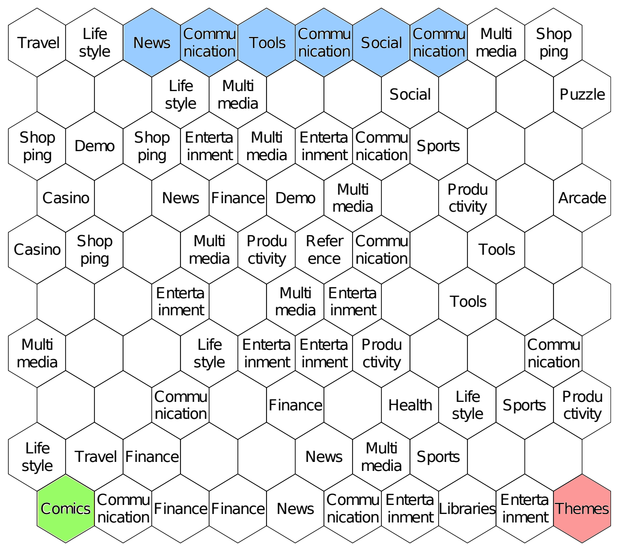

Figure 3: SOM for Android permissions with category labels assigned in a winner-take-all approach

Figure 2 shows that, in general, applications are sparsely populated (i.e., more light regions than dark) in the permission space with two exceptions: the dark shaded regions in the lower right and left corners that correspond to applications from Comics and Themes respectively (shaded in Figure 3). Themes is a general category which contains user interface enhancements to both the Android OS and its applications. Sparsely populated regions (i.e., light regions in Figure 2) involve the applications which vary substantially in terms of requested permissions.

In Figure 3, we see that the Communication category populates the upper middle section of the SOM, although it is mixed with applications from categories such as News, Tools, Social (shaded upper region, in Figure 3). We note that in terms of requested permissions, Communication is the most diverse category, as shown in Table 1. Therefore, applications in this category implement a diverse set of functionality, which consequently causes the Communication category to represented by numerous neurons of the SOM.

Further inspection revealed that the most prevalent types of applications in the upper middle region were applications that make use of network communications and standard phone features (i.e., making and receiving calls, sending and receiving text messages). Given that SOM places similar input patterns in the same region, this indicates that applications from the same category do not necessarily 'behave' similarly (i.e., do not request similar permissions). Hence, applications requesting similar sets of permissions (from multiple categories) are clustered together which implies that applications from different categories can request similar sets of permissions. This reflects the fact that the categories defined by Google are based on semantic activity classes rather than the technical features used to implement them.

In order to identify the frequency of use and the correlations between the permissions, we expand our analysis by performing *component plane analysis* [16] of the SOM. Component analysis reduces the dimensionality of the input space allowing us to visualize the use of permissions in a 2-dimensional space. Compared to Figures 2 and 3, component plane analysis in Section 5.1 summarizes the use of Android permissions in separate visualizations exclusive to the permission in question.

5.1 Component Plane Analysis

Understanding the relationships between variables in complex data is a challenging task. The 1,100 Android applications in our dataset requested access to 119 distinct permissions where each permission represents an additional dimension within which applications are expressed. Component plane visualizations provided in this section represent the component distribution in individual dimensions. In our work, given that each dimension represents a permission, the component plane analysis and visualizations can be considered as a sliced visualization of the SOM. The component plane analysis reported in this paper is specific to the 1,100 applications in the training set. However, we believe that employing the most popular 50 applications from each category provides a representative dataset upon which an empirical analysis of the Android permission model can be performed.

Figure 4 shows the component plane analysis of all the permissions encountered in our dataset. Each component plane in Figure 4 has the relative distribution of a permission. In this representation, light shades represent the frequent use of the permission whereas the dark shades represent the infrequent use. In this section, we highlight some of the interesting results from the component plane visualizations.

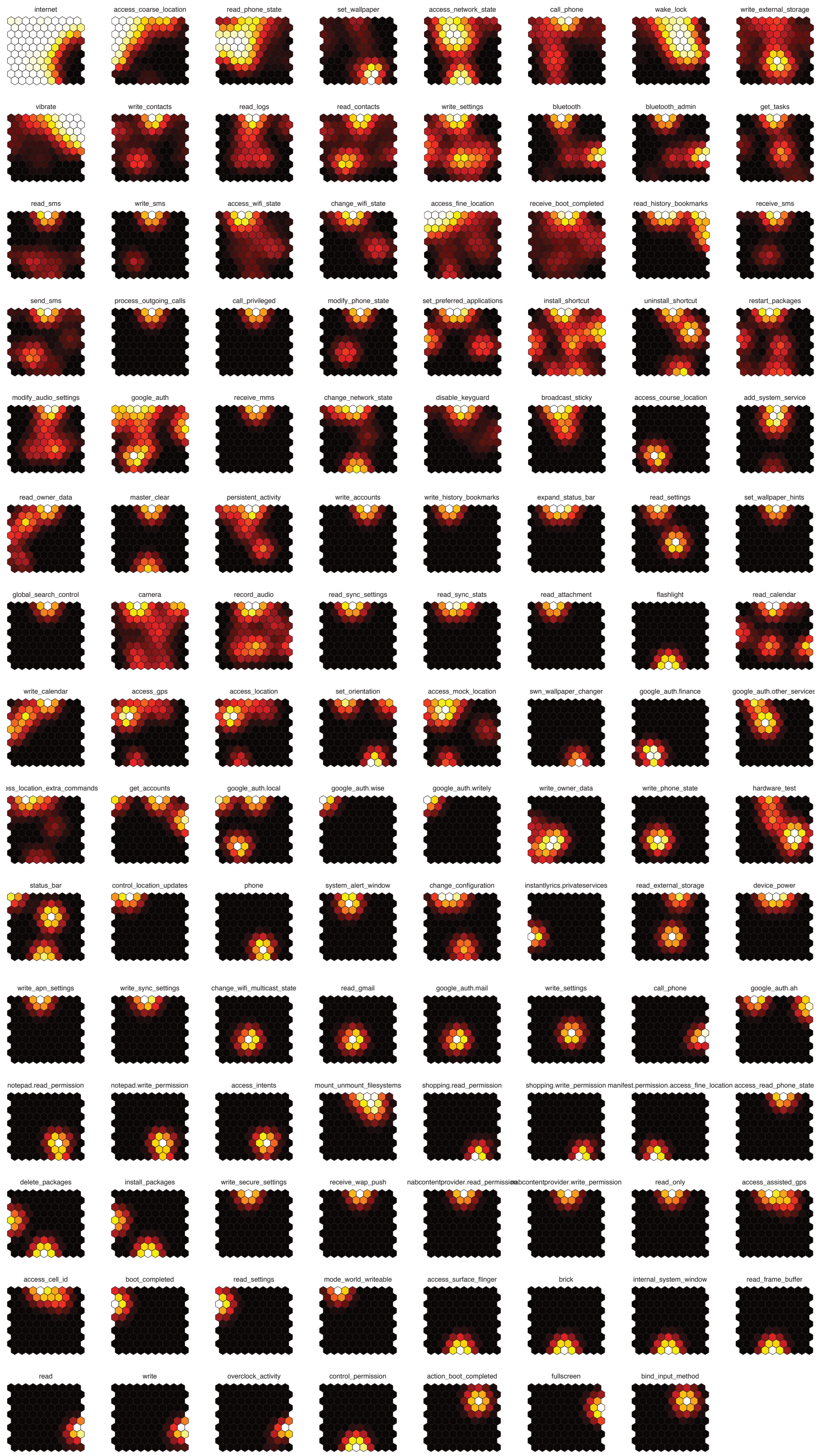

Figure 4: Component plane analysis of the permissions. The lighter areas indicate the regions in which the permission was requested (see Section 5.1).

Figure 5 shows the component plane for the a.p.INTERNET permission. Light shades indicate the regions in which the permission is requested. The component analysis indicates that the a.p.INTERNET permission covers a large portion of the map which means it is requested by the majority of applications in our dataset (over 60%). The lower right region in Figure 5 represents the few applications which do not request the a.p.INTERNET permission, mainly the applications in the Theme and Productivity categories (based on labels in Figure 3). Smartphones strive for always-on Internet connectivity, and can attain this by providing different connection modes; 3G, Wi-Fi and Bluetooth are ways in which the Android OS (and most other smartphones) can connect to online services. Our dataset also contained only free applications which in many cases are ad-supported. Advertisements are pulled from the Internet causing developers to request this permission, even if their application requires no connectivity for its core functionality.

Furthermore, smartphones, and Android in particular have a strong focus on 'cloud' services, which are hosted on a remote server. Many of the Android applications are tightly integrated with Google servers. This also makes sense for devices with low computing power.

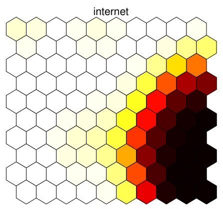

Figure 5: Component plane visualization of the a.p.INTERNET permission

The analysis of component planes can reveal correlations between permissions. Specifically, two permissions are likely to be correlated, if the component plane visualizations are similar. Figure 6 shows the component planes for the location-related permissions, providing location information at varying degrees of accuracy. The component plane visualizations of a.p.ACCESS_COARSE_LOCATION and a.p.ACCESS_FINE_LOCATION in Figure 6 show that both permissions are requested on the upper right region of the map (hexagons in the same region are light shaded). This region corresponds to the Travel, Shopping, Communication and Lifestyle categories, in Figure 3. Applications requiring access to fine location are slightly more biased towards the upper right corner, further narrowing down that area to applications that need precise location information, such as navigation applications. Furthermore, given that the component visualizations of a.p.ACCESS_ASSISTED_GPS and a.p.ACCESS_CELL_ID are very similar in Figure 6, they are highly correlated. This is expected since assisted GPS relies on the cell tower ID for location.

Our analysis determined that pairs of permissions are common (Figure 4), such as requesting both a.p.ACCESS_COARSE_LOCATION and a.p.ACCESS_FINE_LOCATION. We believe that the location permission was divided in this way by Google with the hope that developers would use coarse location for services like news or weather applications (which do not need to know your exact GPS coordinates, but simply a general geographic area). Fine location would be used by navigation applications. However, we see developers requesting both coarse and fine location frequently (sometimes even mock location, which creates a 'simulated' location for testing purposes).

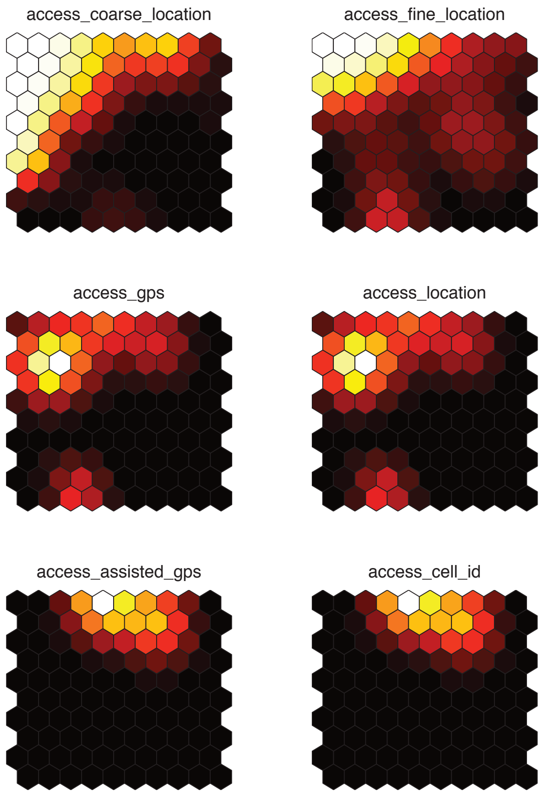

Figure 6: Component plane visualization of the location related permissions

In addition to location-aware applications, various applications such as tools and messaging apps commonly use pairs of permissions. For example, Figure 4 demonstrates that the a.p.WRITE_SMS and a.p.READ_SMS permissions occur in the same region. This implies that applications requesting SMS permissions also request similar sets of permissions, which resulted in both permissions to be active in the same region. Moreover, various other permission pairs are observed such as writing to and reading from the external storage and installing and uninstalling shortcuts. Enck et al. [13] note that applications that have both the send and write or the receive and write SMS permission labels are 'dangerous' due to being able to intercept or transmit messages without the user's knowledge. It is important to note that although these permissions are correlated in our component plane analysis, this does not mean that applications are requesting both permissions.

Rather, this means that applications have similar characteristics if they are requesting send or receive SMS functionality. This could lead to false positives if classifying malware based only on component plane analysis.

The component analysis of the permissions related to system settings are shown in Figure 7. Applications in Tools and Communication categories (shaded upper region in Figure 3) require access to system settings hence the light shaded hexagons in the corresponding upper region.

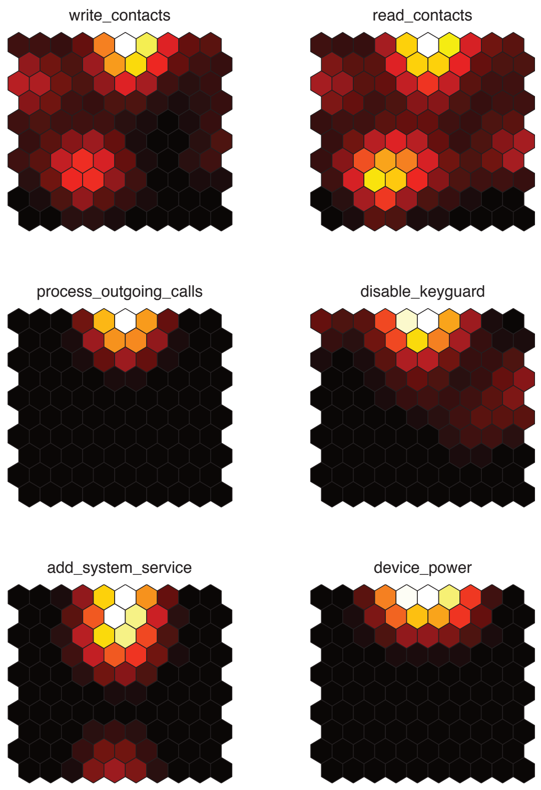

Figure 7: Component plane visualization of the permissions related to system settings

Figure 8 shows the commonly requested permissions which are used by applications in different regions of the SOM (light shaded throughout the SOM). These permissions correspond to common tasks such as installing shortcuts, using the camera and recording audio. For example, the a.p.CAMERA permission is in the Shopping category (applications that allow the user to take a picture of a barcode for price comparisons) as well as Communication (using the camera for sending pictures or videos), Travel (taking pictures of landmarks or tourist attractions), etc. Commonly requested permissions tend to be scattered all over the SOM.

In addition to a small set of permissions requested by a substantial number of applications, many permissions are requested by only a few applications. We believe that these infrequent permissions are mainly defined by developers to facilitate interaction with other Android applications. On the other hand, frequently requested permissions, particularly a.p.INTERNET do not provide the sufficient ex-

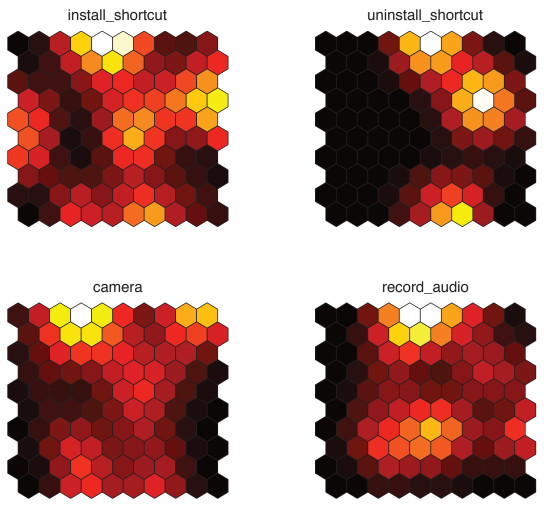

Figure 8: Component plane visualization of the commonly requested permissions

pressiveness to enforce control over the degree of Internet access that the application obtains. Thus, we believe that a.p.INTERNET permission fails to provide sufficiently fine grained control of the resources. By contrast, there exist numerous developer-defined permissions which effectively act as an ‘access control list exception’, allowing other applications to access functionality through a newly defined API. In other words, they specify which other applications can communicate with the application in question rather than a permission, which specifies the resources that the application in question can access. We suggest that the current Android permission model can be improved by further distinguishing between different classes of Internet access, while providing a mechanism for developers to specify an access control list without the use of self-defined permissions.

6. FURTHER DISCUSSION

Designing a permission-based system is a challenging task because system designers must anticipate what usage will be given to the permissions defined in their system. The analysis in this paper has helped to identify developer usage patterns in a real-world dataset of top Android applications. Additionally, there is a constant struggle to make the system highly configurable under different use-cases while maintaining a low level of complexity. Understanding how the permission model is used in practice can help in making modifications to improve currently deployed permission systems.

Our analysis shows that in Android a small number permissions are very frequently used and a large number of permissions are only occasionally used. Furthermore, our analysis shows correlations between several of the infrequently used permissions.

We note that having finer-grained permissions in a permission-based system enables users to have detailed control over what actions are allowed to take place.

Whether it is beneficial to provide finer granularity will depend on many factors within a particular environment, as it increases complexity and thus may have, for example, usability impacts on designers and end-users.

In the case of Android, having 'too many' permissions impacts both developers and end users. Developers must understand which permissions are needed to perform certain actions; determining this is often non-trivial, even for 'experts'. While some enthusiastic developers might take the time to learn what each of the 110 or more permissions do and request them appropriately when needed, other developers might choose to simply over-request functionality to make sure their application works. Smartphone users themselves are (currently, through no choice of their own) heavily involved in the Android permission architecture since it is they who either allow or deny the installation of applications based on the presented set of requested permissions. Whether or not this is the best approach to handling such a complex permission system is beyond our present scope, but does lead us to question if it is reasonable to ask non-technical users to configure these complex systems.

6.1 Possible Enhancements to Android

The Android permission model does not currently make use of the implied hierarchy in its namespace. For example, `a.p.SEND_SMS` and `a.p.WRITE_SMS` are two independent permission labels, instead of being grouped, for instance, under `a.p.SMS.*`. Android includes an optional logical permission grouping [9] that is used for displaying permissions with more understandable names (e.g., one of the groupings reads "Services that cost you money" instead of `a.p.CALL_PHONE`). This grouping, however, does not allow developers to hierarchically define permissions, which could potentially extend current Android-defined permissions to express more detailed functionality.

In the case of Android particularly, a permission hierarchy would allow for an extensible naming convention and help developers more accurately select (only the) needed features. One example would be a free application that displays ads from domains belonging to Admob. Currently a developer would include the ad code snippet, and request the `a.p.INTERNET` permission. This permission allows the application to communicate over any network and retrieve any data from any server in the world. A more fine grained hierarchical permission scheme could enable the developer to request the `a.p.INTERNET.ADVERTISING(*.admob.com)` permission which could limit network connectivity to only download ads in static HTML from subdomains of Admob. A hierarchical permission scheme could help users understand why an application is requesting specific permissions, but more importantly, could help developers better use the principle of least privilege. This modification is not backwards compatible with the currently deployed Android OS, therefore it might be better suited for an entirely new model instead.

Developers that make use of self-declared permissions to restrict access to APIs in their third party applications add complexity to the system with every new permission defined. In the current flat permission model, new developer-introduced permissions will likely be used infrequently and independently of other permissions. Logically grouping all self-defined permissions under one category could help inform users that these permissions are not part of the core (Google-defined) set.

Another possible enhancement to Android is to identify permissions that when used independently, do not pose a serious threat to the device or the user. These permissions could be displayed in one informative-only collection prior to prompting the user to (as currently done) individually accept other, perhaps more high-risk permissions. This may significantly reduce the number of individual acceptance decisions presented to the user, and hopefully increase user attention across the remaining items.

6.2 Applicability to Other Permission-Based Systems

The methodology presented in this work has allowed us to understand how developers use the permission-based security model in Android. We believe that our methodology is applicable to explore usage trends in other permission based-based systems. A base requirement for the methodology to work is being able to display applications and associated permissions as a bit string as described in Section 3.3. For this representation to be possible, the set of permissions requested by an application must be accessible. In the case of Android, the set is statically readable in a manifest, but other systems might have different implementations.

Google's Chrome OS extension system [4, 10] uses an Android-like manifest and permissions to access advanced functionality, which makes this system a prime candidate for applying our methodology. An empirical study of a large set of third-party extensions using our SOM-based methodology, could help identify what correlations, if any, are present in requesting permissions to open tabs, read bookmarks, etc. This may also be of use in addressing other security concerns raised in recent work [10].

7. CONCLUSION

We have introduced a methodology to the security community for the empirical analysis of permission-based security models. In particular, we analyzed the Android permission model to investigate how it is used in practice and to determine its strengths and weaknesses. The Self-Organizing Map (SOM) algorithm is employed, which allows for a 2-dimensional visualization of highly dimensional data. SOM also supports component planes analysis which can reveal interesting usage patterns.

We have analyzed the use of Android permissions in a real-world dataset of 1,100 applications, focusing on the top 50 application from 22 categories in the Android market.

The results show that a small subset of the permissions are used very frequently where a large subset of permissions were used by very few applications. We suggest that the frequently used permissions, specifically `a.p.INTERNET`, do not provide sufficient expressiveness and hence may benefit from being divided into sub-categories, perhaps in a hierarchical manner as explained in Section 6.1. Conversely, infrequent permissions such as the self-defined and the complementary permissions (e.g., install/uninstall) could be collapsed into a general category. Providing finer granularity for frequent permissions and combining the infrequent permissions can enhance the expressiveness of the permission model without increasing the complexity (i.e., maintaining a constant overall permission count) as a result of the additional permissions. We hope that our SOM-based methodology, including visualization, is of use to others exploring independent permission-based models.

Acknowledgements

We thank William Enck and members of Penn. State's SIIS lab for making available the dataset used for our analysis. We also thank our CCS shepherd, Patrick McDaniel, and anonymous referees for their helpful comments. The third author acknowledges NSERC funding under a Discovery Grant and as Canada Research Chair in Authentication and Computer Security. Partial funding from NSERC ISSNet is also acknowledged.

8. REFERENCES

[1] Android. `http://www.android.com` Retrieved February 6th, 2010.

[2] Android Market Statistics from Androlib. `http://www.androlib.com/appstats.aspx` Retrieved July 7th, 2010.

[3] BlackBerry APIs with controlled access. `http://docs.blackberry.com/en/developers/deliverables/5580/Java_APIs_with_controlled_access_447163_11.jsp` Retrieved April 9th, 2010.

[4] Formats: Manifest Files - Google Chrome Extensions - Google Code. `http://code.google.com/chrome/extensions/manifest.html#permissions` Retrieved April 9th, 2010.

[5] How Android Security Stacks Up. `http://www.technologyreview.com/communications/24944/page1/` April 1st, 2010.

[6] Independent Security Evaluators - Exploiting Android. `http://securityevaluators.com/content/case-studies/android/` Retrieved January 15th, 2010.

[7] The Android Developer's Guide. `http://developer.android.com/guide/index.html` Retrieved January 29th, 2010.

[8] The Android Developer's Guide - Android Manifest Permissions. `http://developer.android.com/reference/android/Manifest.permission.html` Retrieved April 5th, 2010.

[9] The Android Developer's Guide - Permission Groups. `http://developer.android.com/guide/topics/manifest/permission-group-element.html` Retrieved April 7th, 2010.

[10] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*.

[11] K. Beznosov, P. Inglesant, J. Lobo, R. Reeder, and M. E. Zurko. Usability meets access control: challenges and research opportunities. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 73–74, New York, NY, USA, 2009. ACM.

[12] D. Curry. *UNIX System Security*. Addison-Wesley, 1992.

[13] W. Enck, M. Ongtang, and P. D. McDaniel. On Lightweight Mobile Phone Application Certification. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 235–245. ACM, 2009.

[14] W. Enck, M. Ongtang, and P. D. McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[15] J. Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[16] T. Kohonen. *Self Organizing Maps*. Springer, third edition, 2001.

[17] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[18] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel. Semantically rich application-centric security in android. In *ACSAC*, pages 340–349. IEEE Computer Society, 2009.

[19] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong. Expandable grids for visualizing and authoring computer security policies. In *CHI '08*, pages 1473–1482, New York, NY, USA, 2008. ACM.

[20] D. K. Smetters and N. Good. How users use access control. In *SOUPS '09: Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 1–12, New York, NY, USA, 2009. ACM.

[21] A. Ultsch and H. Siemon. Kohonen's self-organizing feature maps for exploratory data analysis. In *Proceedings of the International Neural Network Conference (INNC'90), Dordrecht, Netherlands*, pages 305–308. Kluwer, 1990.

[22] J. Vesanto. *Data Mining Techniques Based on the Self-Organizing Map*. Master's Thesis, Helsinki University of Technology, May 1997.